Python Typing Module Tutorial

What is the Typing Module?

The `typing` module provides support for type hints in Python, allowing you to specify the expected types of variables, function parameters, and return values. This makes your code more self-documenting and helps catch type-related errors early.

When to Use Type Hints

1. Function Parameters and Return Types

Always use type hints for function parameters and return values, especially in:

- Public APIs
- Complex functions
- Functions used by multiple modules
- When the type isn't obvious from the function name

python

```python
from typing import List, Dict, Optional, Union

def process_data(items: List[str], config: Dict[str, int]) -> Optional[str]:
    """Process a list of items with given configuration."""
    if not items:
        return None
    return items[0] * config.get('multiplier', 1)
```

2. Class Attributes and Methods

Use type hints for class attributes and methods to clarify the expected data structure:

```python
from typing import ClassVar, List, Optional

class DataProcessor:
    # Class variable
    default_config: ClassVar[Dict[str, int]] = {'batch_size': 100}

    def __init__(self, name: str, items: List[str]) -> None:
        self.name: str = name
        self.items: List[str] = items
        self.processed_count: int = 0

    def get_item(self, index: int) -> Optional[str]:
        return self.items[index] if 0 <= index < len(self.items) else None
```

3. Complex Data Structures

When working with nested data structures, collections, or custom types:

```python
from typing import Dict, List, Tuple, Set, Union

# Nested structures
UserData = Dict[str, Union[str, int, List[str]]]
Coordinates = Tuple[float, float]
ProcessingResult = Tuple[List[str], Dict[str, int], bool]

def analyze_users(users: List[UserData]) -> ProcessingResult:
    """Analyze user data and return results."""
    names = [user['name'] for user in users if isinstance(user['name'], str)]
    stats = {'total': len(users), 'active': sum(1 for u in users if u.get('active'))}
    success = len(names) > 0
    return names, stats, success
```

Common Typing Constructs

Basic Types

python

```python
# Built-in types
name: str = "Alice"
age: int = 30
height: float = 5.8
is_active: bool = True
```

Collections

python

```python
from typing import List, Dict, Set, Tuple

# Lists
numbers: List[int] = [1, 2, 3, 4]
mixed_list: List[Union[str, int]] = ["hello", 42, "world"]

# Dictionaries
user_ages: Dict[str, int] = {"alice": 30, "bob": 25}
complex_dict: Dict[str, List[int]] = {"scores": [85, 92, 78]}

# Sets
unique_ids: Set[str] = {"id1", "id2", "id3"}

# Tuples
coordinates: Tuple[float, float] = (10.5, 20.3)
variable_tuple: Tuple[str, ...] = ("a", "b", "c", "d")  # Variable length
```

Optional and Union Types

python

```python
from typing import Optional, Union

# Optional (can be None)
def find_user(user_id: str) -> Optional[Dict[str, str]]:
    # Returns user dict or None if not found
    pass

# Union (can be one of several types)
def process_id(identifier: Union[str, int]) -> str:
    return str(identifier)
```

Callable Types

python

```python
from typing import Callable

# Function that takes two ints and returns int
Calculator = Callable[[int, int], int]

def add(a: int, b: int) -> int:
    return a + b

def apply_operation(x: int, y: int, operation: Calculator) -> int:
    return operation(x, y)

result = apply_operation(5, 3, add)  # result = 8
```

Generic Types

python

```python
from typing import TypeVar, Generic, List

T = TypeVar('T')

class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: List[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

# Usage
int_stack = Stack[int]()
str_stack = Stack[str]()
```

Advanced Typing Features

Type Aliases

python

```python
from typing import Dict, List, Union

# Create readable aliases for complex types
JSON = Union[Dict[str, 'JSON'], List['JSON'], str, int, float, bool, None]
UserId = str
UserProfile = Dict[str, Union[str, int, List[str]]]

def process_json(data: JSON) -> None:
    pass

def get_user_profile(user_id: UserId) -> UserProfile:
    pass
```

Protocols (Structural Typing)

python

```python
from typing import Protocol

class Drawable(Protocol):
    def draw(self) -> None: ...

class Circle:
    def draw(self) -> None:
        print("Drawing circle")

class Square:
    def draw(self) -> None:
        print("Drawing square")

def render_shape(shape: Drawable) -> None:
    shape.draw()  # Works with any object that has a draw() method
```

Literal Types

python

```python
from typing import Literal

Mode = Literal['read', 'write', 'append']

def open_file(filename: str, mode: Mode) -> None:
    # mode can only be 'read', 'write', or 'append'
    pass
```

Best Practices

1. Start Simple

Begin with basic type hints and gradually add more complex ones:

python

```python
# Start with this
def greet(name: str) -> str:
    return f"Hello, {name}!"

# Then progress to this
def process_users(users: List[Dict[str, Union[str, int]]]) -> Dict[str, List[str]]:
    pass
```

2. Use Type Aliases for Readability

python

```python
# Instead of this
def process_data(data: Dict[str, List[Tuple[str, int, bool]]]) -> List[Dict[str, Union[str, int]]]:
    pass

# Use this
DataRecord = Tuple[str, int, bool]
InputData = Dict[str, List[DataRecord]]
OutputRecord = Dict[str, Union[str, int]]

def process_data(data: InputData) -> List[OutputRecord]:
    pass
```

3. Don't Over-Type

Avoid excessive type annotations for simple, obvious cases:

python

```python
# Probably unnecessary
x: int = 5
items: List[str] = ["a", "b", "c"]

# Better - let Python infer these
x = 5
items = ["a", "b", "c"]
```

When NOT to Use Type Hints

1. Simple Scripts

For quick, one-off scripts where types are obvious and the code won't be maintained long-term.

2. Performance-Critical Code

Type hints add minimal runtime overhead, but in extremely performance-sensitive code, you might skip them.

3. Highly Dynamic Code

When working with highly dynamic code where types change frequently, type hints might be counterproductive.

Tools and Integration

Type Checkers

- **mypy**: The most popular static type checker
- **pyright**: Microsoft's type checker (used by Pylance)
- **pyre**: Facebook's type checker

IDE Integration

Most modern IDEs (PyCharm, VSCode, etc.) provide excellent support for type hints:

- Auto-completion
- Error detection
- Refactoring assistance
- Better code navigation

Runtime Type Checking

```python
from typing import runtime_checkable, Protocol

@runtime_checkable
class Serializable(Protocol):
    def serialize(self) -> str: ...

class MyClass:
    def serialize(self) -> str:
        return "serialized"

obj = MyClass()
if isinstance(obj, Serializable):  # Works at runtime
    print(obj.serialize())
```

Summary

Use type hints when:

- Writing functions that will be used by others
- Working with complex data structures
- Building maintainable, long-term codebases
- You want better IDE support and error detection
- The types aren't immediately obvious from the code

The typing module makes Python code more robust, readable, and maintainable while preserving Python's dynamic nature. Start with simple type hints and gradually incorporate more advanced features as your needs grow.